Top-Down Composition of Software Architectures

de Bruin, H.; van Vliet, H.

published in
Proceedings 9th International Conference on the Engineering of Computer-Bsed Systems (ECBS)
2002

document version
Publisher's PDF, also known as Version of record

Top-Down Composition of Software Architectures

Hans de Bruin Hans van Vliet
Vrije Universiteit, Amsterdam
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
e-mail: {hansdb,hans}@cs.vu.nl

Abstract

This paper discusses an approach for top-down composition of software architectures. First, an architecture is derived that addresses functional requirements only. This architecture contains a number of variability points which are next filled in to address quality concerns. The quality requirements and associated architectural solution fragments are captured in a so-called Feature-Solution (FS) graph. The solution fragments captured in this graph are used to iteratively compose an architecture. Our versatile composition technique allows for pre- and post-refinements, and refinements that involve multiple variability points. In addition, the usage of the FS graph supports Aspect-Oriented Programming (AOP) at the architecture level.

1 Introduction

The architecture of a software system captures early design decisions. These early design decisions reflect major quality concerns, including functionality. In order not to reinvent the wheel time and again, we would like to capture chunks of architectural knowledge explicitly, and use these chunks when deriving an architecture that fulfills some set of quality concerns. Our solution for capturing this knowledge is a Feature-Solution (FS) graph, which connects quality requirements with solution fragments at the architectural level [4]. The present paper is concerned with composing an architecture out of the solution fragments captured in the FS-graph.

The process we envisage for deriving the architecture is an iterative, quality-driven approach to software architecting (see for instance [1]). The first step in this process is the derivation of a software architecture that meets the functional requirements set. This is called the reference architecture. Next, the attention focuses on non-functional requirements by iteratively applying known design solutions (e.g., architectural and design patterns) to refine the software architecture. Typically, this requires several iterations. These iterations might also involve backtracking steps because we usually have to deal with conflicting requirements.

At the heart of this iterative, quality-driven process for composing software architectures is the Feature-Solution (FS) graph. In this FS graph, requirements are connected with design solutions. On the basis of requirements specified in the feature space of the FS graph, solutions that are likely to meet the requirements are selected in the solution space. One way to look at the FS graph is that it embodies domain knowledge (expressed as requirements) along with design solutions for solving particular problems in that domain. The role of the FS graph in this iterative approach has been discussed extensively in [4].

In this paper we focus on a composition technique to systematically derive a software architecture by recursively applying design solutions to a given reference architecture. The reference architecture contains variability points at those places where the architecture is expected to be varied, for instance, to cater for non-functional requirements. The variations (e.g., design solutions in the form of patterns) in their turn may contain variability points as well. So in principle, the architecture can be refined indefinitely.

We call our approach a top-down approach to software architecture composition to contrast it with the (bottom-up) composition and (top-down) decomposition approaches that are well-known within software engineering, even though this may sound as a contradiction in terms. In a bottom-up composition, we start with a set of elementary components and aggregate these into higher level components using some glue mechanism. The abstraction level is raised each time we compose a component out of subcomponents. In a system decomposition, the opposite direction is followed. We then start with a top level view of the system and recursively decompose

it into manageable pieces. Thus, system decomposition lowers the abstraction level. In our approach, the starting point is a reference architecture, a description of basic functionality with variability points. The variability points are next used to plug-in existing design solutions that address certain quality concerns. These plug-ins in turn may contain variability points as well which are next used to further fine-tune the solution. This leads to a top-down approach of system composition.

The FS graph plays a key role in the top-down, iterative composition process. Because the FS graph is supposed to contain relevant domain knowledge for system construction, it is also used to explore design alternatives that have a system-wide impact. For instance, it is quite possible to change the interaction style of a system from user-centered (i.e., the user is in control) to system-centered (i.e., don't call the system, the system will call you). This can be seen as the architecture-level counterpart of aspect weaving as used in aspect-oriented programming languages [6].

Use Case Maps (UCM)[1] [2, 3] are used in this paper as a vehicle to demonstrate the principles. UCM is a diagrammatic modeling technique to describe behavioral and, to a lesser extent structural, aspects of a system at a high level of abstraction. UCM provides stubs (i.e., the hooks or variability points) where the behavior of a system can be varied statically at construction time as well as dynamically at run time. The advantage of UCM is that it focuses on the larger, architectural issues, and its support of plug-ins and stubs, both static and dynamic. However, the use of UCM is not a prerequisite; the composition technique can also be applied using, for instance, UML.

The contributions of this paper are twofold. Firstly, we present a versatile composition technique for software architectures. This technique offers pre- and post refinement, supports multiple plug-ins in a design solution, and offers rules to enforce the well-formedness of refinements. Secondly, the usage of the FS graph supports abstraction level lowering and Aspect-Oriented Programming (AOP) at the architecture level.

The remainder of this paper is organized as follows. Section 2 explains our technique for top-down composition. In section 3, we illustrate the role of the FS graph in system composition. Section 4 illustrates the use of the FS-graph to realize global transformations to a software architecture. Section 5 discusses related work, and section 6 contains our conclusions.

2 Composition Technique

In [4], we described a very simple, but effective way of system composition through successive refinements. The basic idea is to provide a UCM with stubs in which plug-ins can be placed. A plug-in may contain a stub as well, so a plug-in can be placed in a plug-in, and so on. This approach was used, amongst others, to secure the communication between WEB-browsers and a WEB-based system by first adding encryption/decryption components and next a firewall component.

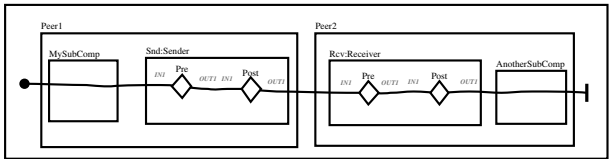
This idea works out fine for local refinements, that is, refinements such as encryption and firewall that apply to one variability point (UCM stub) only. It falls short when we have to deal with refinements that should be applied to multiple variability points. An example of the latter type of refinement is the use of the Observer design pattern, which affects both a component in the role of subject and one or more components in the role of observers. The solution presented below can handle refinement of multiple variability points.

The basic concepts of refinement are explained using the example shown in Figure 1. This first example again deals with encryption/decryption and firewall refinement. This time, though, we model encryption and decryption as one refinement with two variability points, which gives greater flexibility than the solution presented in [4]. The example involves two components, Peer1 and Peer2. Component Peer1 communicates with component Peer2. Peer1 provides a socket for sending data, whereas Peer2 provides a socket for receiving data. The communication between the peers is refined by first adding encryption and decryption components in Peer1 and Peer2, respectively, and next a firewall component in Peer2. The meaning of the UCM elements used for refinement is given in Table 1. Notice that we have two options for firewall placement in this example. Either the firewall component is placed before the Decryptor component, or it is placed after it, depending on whether the firewall is placed in the Pre-stub or in the Post-stub of the Rcv (Receiver) socket in Peer2, respectively.

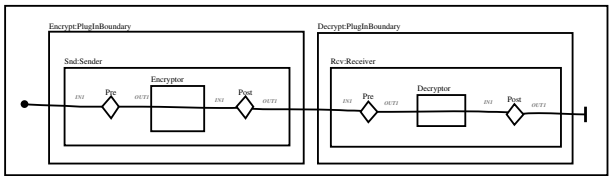
Refinements have to obey certain rules. These rules enforce the well-formedness of the transformation between successive architectures. These rules are easily understood by viewing a refinement and the UCM to which it is applied as patterns that should match. The rules for refinement are as follows:

[1] For readers not familiar with UCM, a short introduction can be found in appendix A.

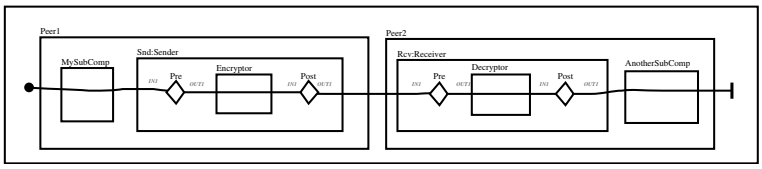
A basic UCM modeling peer to peer communication in which each peer provides a socket for refinement:



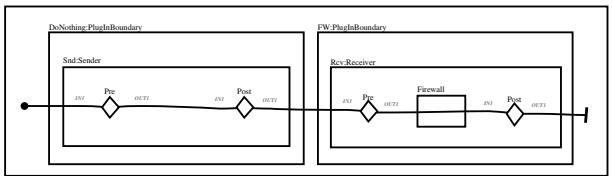
A refinement containing a plug-in for encryption and decryption:



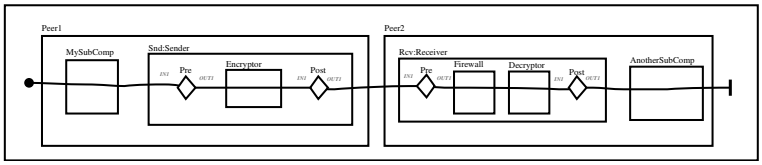
The result of refining the basic UCM with encryption and decryption:



A refinement containing a firewall plug-in:



The result of placing the firewall refinement in the Pre-stub:



The result of placing the firewall refinement in the Post-stub:

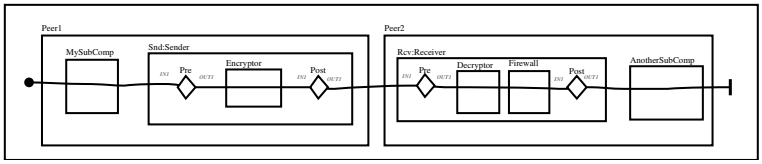


Figure 1: Refinement basics.

Type matching. Sockets and plug-ins have names and type identifiers, specified as a Name:Type pair. Types are organized in type hierarchies supporting single as well as multiple inheritance. A plug-in can be placed in a stub of a socket only if the plug-in and socket have matching types. That is, the type of a plug-in must be identical to or be a subtype of the stub's type.

Element	Meaning
Socket and Stubs	A socket is a placeholder that provides at most two (UCM) stubs in which a (UCM) plug-in can be placed. One stub is called Pre, the other one Post. The intention is that the Pre-stub provides the ability for pre-processing. Likewise, the Post-stub offers the possibility for post-processing. Stubs are depicted as diamonds.
Plug-In	A plug-in can be placed in either the Pre-stub or the Post-stub of a socket. It may itself contain a Pre-stub and Post-stub, thereby creating the opportunity to recursively plug in plug-ins. If a plug-in is placed in the Pre-stub, the Post-stub of the plug-in is discarded, and vice-versa for Post-stub placement.
Plug-In-Boundary	A plug-in-boundary encloses a set of plug-ins that should be placed in a single component. It is used as a mechanism to ensure that plug-ins that belong together are not scattered over a number of components. A plug-in boundary is depicted as a rectangle surrounding its plug-ins.
Refinement	A refinement is a combination of plug-ins enclosed in one or more plug-in boundaries that can be placed in stubs provided by sockets.

Table 1: The meaning of UCM refinement elements.

Plug-in matching. Three kinds of plug-ins are recognized:

Mandatory. All mandatory plug-ins in a refinement must be placeable in sockets with matching types.

Optional. Only if a socket is provided for an optional plug-in, the plug-in will be placed in the socket, otherwise it is discarded.

Extension. In contrast to an optional plug-in, no socket needs to be provided, i.e., an extension is never discarded.

The plug-in kind is specified syntactically as Name:Type (Plug-in kind)$_{optional}$. If no plug-in kind is specified, the plug-in is qualified as mandatory.

Boundary matching. All plug-ins enclosed in a plug-in boundary must be placed in one component.

Structure matching. If a UCM path (indicated by a straight or curved line) connects two components, that means that those components communicate with each other, in some way or another. There then exists a structural relationship between those components. There are two ways to specify further details about this relationship: the communication may be one-to-one or one-to-many, and the communication may be direct or indirect. Since components reside within plug-in boundaries, we may also say that these relationships exist between plug-in boundaries.

The structural relations between plug-in boundaries in a refinement must correspond with the structural relations provided by sockets.

A path to a stack of components denotes a one-to-many relation. For instance, in the Observer design pattern, a stack of observers may be connected to a single subject. A structural one-to-many relation in a refinement matches a one-to-one relation in the target UCM, but not vice versa.

A distinction is further made between indirectly and directly connected plug-ins in a refinement. In case of indirectly connected plug-ins (as indicated by an UCM end symbol attached to a start symbol), the corresponding sockets need not necessarily have a direct connection. All that is required is that there is a UCM path from one socket to the other one, i.e., the path may go through an arbitrary number of components and sockets. The latter is not permitted for direct connections.

To illustrate the finer details of the refinement process, consider again the example of peer to peer communication. In the example in Figure 1, we have abstracted away from how the peers communicate. One way of realizing this is by deploying the Observer design pattern, which can be captured in a UCM refinement as shown in Figure 2.

Figure 3 shows how the Observer refinement can be plugged in the peer to peer UCM with encryption/decryption and firewall components. Firstly, notice that the types specify where plug-ins are placed. The plug-in of type Sender fits into Peer1, while the plug-in of type Receiver fits into Peer2. Both these plug-ins are mandatory. There

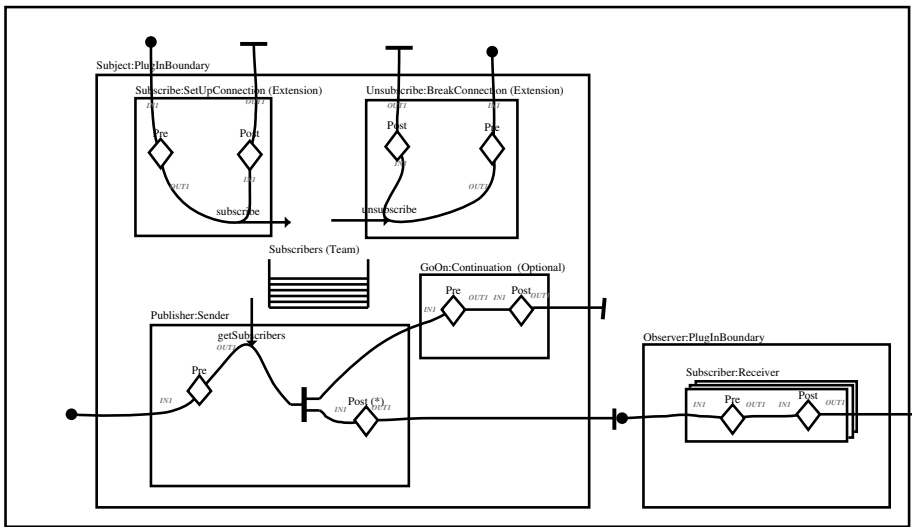


Figure 2: Observer design pattern captured in a UCM refinement.

is an optional continuation plug-in for which no socket is provided, so that one is discarded in the refinement. There are two extension plug-ins which are placed into Peer1 (because they are contained within the plug-in boundary that also contains the Publisher:Sender plug-in), which now contains functionality for subscribing and unsubscribing. The structural relationship between the two plug-in boundaries of the Observer refinement are one-to-many and indirect, so it fits the one-to-one direct relationship between the two peers in Figure 1.

A few remarks are in order here. Firstly, by using a specific solution, more and more details are added to the basic UCM we started with in the first place. We call this process abstraction lowering, which is discussed further in the next section. Conversely, the Observer design pattern is captured in a general UCM refinement that can be used in a variety of circumstances. Secondly, a flexible refinement mechanism is provided in the form of Pre-stubs and Post-stubs. In this particular case, the multicast to interested observers can be positioned either before or after the encryption component (in Figure 3, we have chosen to place it before the Encryptor).

protection:

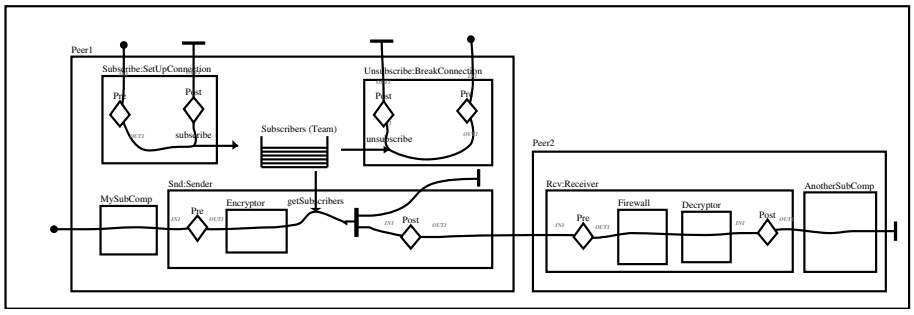


Figure 3: Lowering the abstraction level by deploying the Observer pattern.

3 Abstraction lowering and Refinement

By starting at the top-level system structure, each refinement in the form of plug-in placement lowers the abstraction level. An example was given in the previous section in which the Observer pattern refinement resulted in the extension of the peer to peer system with subscription management. In our approach, the architectural knowledge of these refinements and the quality concerns they address are captured in a Feature-Solution (FS) graph.

As an example, consider again peer to peer communication. A FS graph that details how the basic peer to peer UCM can be refined is given in Figure 4. Two spaces are recognized in the FS graph. The Feature (F) space contains the requirements, whereas the Solution (S) space contains solutions in the form of plug-ins that can be placed in the basic peer to peer UCM. Features as well as solutions are decomposed in AND-(EX)OR decomposition trees. An AND decomposition of a node means that all its constituents must be available, an OR requires an arbitrary (≥ 0) number of constituents, and an EXOR requires precisely one constituent. The key idea is that a feature in the F-space can select a solution in the S-space as defined by directed selection links between nodes (indicated by a solid line). It is also possible to explicitly rule out a particular solution. This is done by connecting a feature with a *negative* selection link to a solution (indicated by a dashed line).

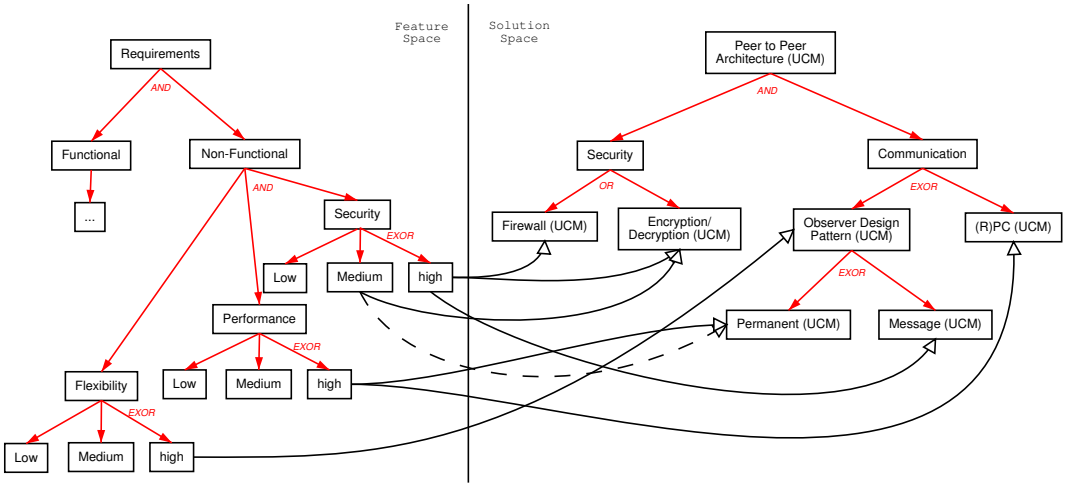


Figure 4: Feature-Solution graph for peer to peer communication.

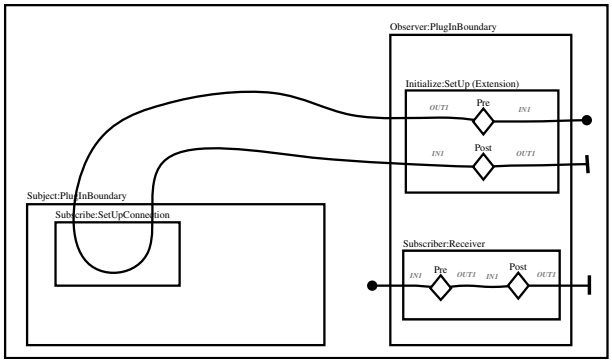
In the example, we focus on non-functional requirements, in particular flexibility, security and performance requirements. If a high flexibility level is desired, the FS graph dictates that we should use the Observer design pattern, because of its properties of reducing the coupling between peers and supporting multiple observers. On the other hand, if we want high performance, the FS graph selects a direct invocation style in the form of (remote) procedure calls. It is interesting to observe that a high level of flexibility and a high level of performance cannot be obtained simultaneously since these requirements select solutions that rule out each other, as implied by the EXOR decomposition of the communication node. Thus, a FS graph contains trade-off information as well. Typically, several design process cycles are required to arrive at an architectural design that satisfies all non-functional requirements. This quality-driven, iterative approach to architecture generation and evaluation is described in more detail in [4].

As discussed before, the abstraction level is lowered each time a refinement step is performed, which generally requires that more and more detail must be added to the general solution we started with. For example, in the case of the application of the Observer design pattern, details must be filled in when and how to subscribe and unsubscribe observers. As can be deduced from Figure 4, the FS graph can contain information about abstraction lowering as well. Suppose we have opted for a high flexibility solution, which resulted in selecting the Observer design pattern. The Observer design pattern node in the FS graph is EXOR decomposed in a Permanent and a Message solution, which stands for setting up a connection permanently or setting up and breaking down a connection for each message, respectively. In case of high performance, a permanent connection is selected, whereas for high security a connection is established on a message basis. Notice that if a medium level of security is chosen, a negative selection link rules out a permanent connection explicitly, and because of the EXOR decomposition relation, this implies the selection of the message solution. In many cases, especially when the required flexibility, security and performance levels are set to low, no particular solution is favored. To put it differently, the FS graph contains degrees of freedom that can be used to explore design alternatives in an iterative design process.

The result of abstraction lowering for a permanent connection is shown in Figure 5. The plug-in for setting up a

permanent connection has two plug-in boundaries. One matches the Subscribe:SetUpConnection socket of Peer1, the other matches the Subscriber:Receiver socket of Peer2. Setting up a permanent connection requires some initialization, which is modeled as an extension in the plug-in. When applying the refinement, this extension is added to the result. Once the connection is established, no additional functionality is needed for the actual communication. This is modeled as an 'empty' plug-in of type Subscriber:Receiver. This empty plug-in is needed to determine where the plug-in is to be applied. When applied, it leaves the existing structure intact.

A plug-in for setting up a permanent connection:



The result of applying the plug-in:

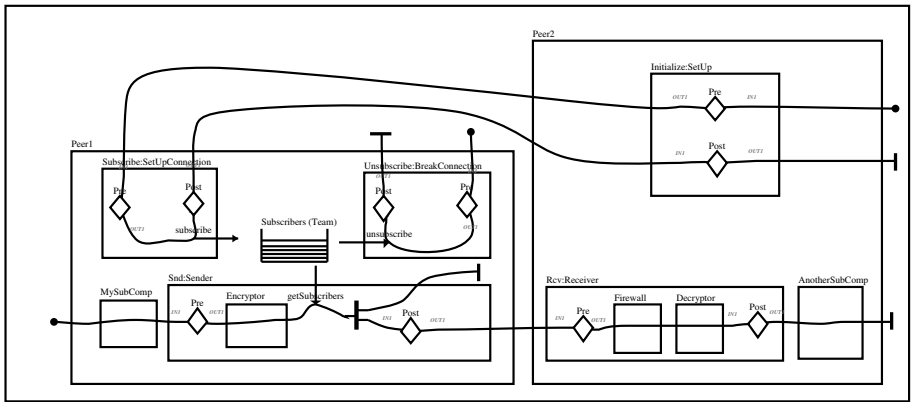


Figure 5: Filling in the details for a permanent connection.

In conclusion, the solution part of the FS-graph thus contains a collection of UCM plug-ins. These plug-ins are used to iteratively refine an architectural solution, guided by the quality requirements as expressed in the feature part of that same graph. Each time we apply such a refinement, the abstraction level of the architecture is lowered.

4 Crosscutting Concerns

By means of a larger example, we show in this section that a FS graph cannot only be used to refine a particular solution locally, but also globally, at the system level. That is, we can handle aspects, such as performance and security, that in general require the adaptation of many components rather than one or a few. This type of refinement may be called Aspect-Oriented Programming (AOP) at the architectural level.

The example is about clients ordering goods by a retailer. A typical but simplified scenario for ordering is as follows. First, a client fills in a form and sends this form electronically to the retailer. Next, the retailer checks its database to see whether the goods are in stock. Finally, it sends a report to the client stating when the goods are expected to be shipped.

Now assume this basic scenario is extended with brokering. If certain goods are not supported by the retailer, he tries to obtain them from wholesalers. Since it can take some time to get answers from wholesalers (especially

if they do not provide on-line services), a client-driven approach as in the basic ordering scenario is no longer applicable. Instead, the retailer takes the initiative to inform the client about the status of an order after all outstanding requests to wholesalers have been collected. Thus, the communication between a client and a retailer has changed from client-driven to a mixture of client-driven and retailer-driven communication.

A frequently used architecture for systems like the ordering system is a Client-Server (CS) architecture. In figure 6, a reference architecture is shown in which the basic scenario of ordering goods is contained. Both the client and the server component provide sockets for further refinement.

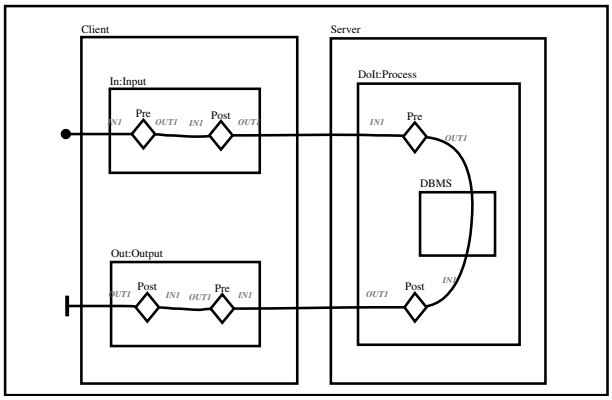


Figure 6: Client-Server reference architecture.

The next step is to extend the system with broker functionality. This is shown in Figure 7. The functionality that is expressed in the UCM is that, first, the database is consulted. If certain goods cannot be delivered by the retailer (i.e., the server), the broker sends a request to a number of wholesalers. Simultaneously, the broker sends a message to the client to inform that brokering is in progress. After all brokering requests have been collected, the database is updated and, finally, a status report is sent to the client. Whereas the basic scenario of ordering goods without brokering is client-driven, the reverse holds for sending a status report after the results of the brokering process have been sorted out. In the latter case, the system takes the initiative. As a consequence, the client has to be adapted to support this system driven approach. For instance, the client can be equipped with an E-mail component or, alternatively, it may contain some kind of message broker. The socket Notify:Messenger contained in the client component acts as a placeholder for system-driven message delivery.

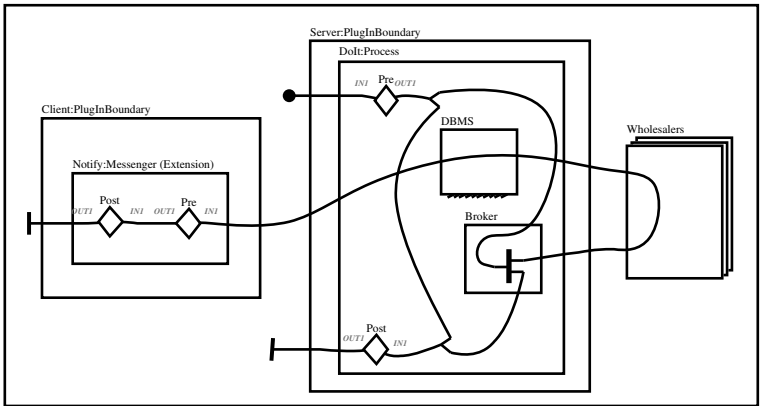
The point is that, because we extend the server with broker functionality, we have to extend the client as well. So the effect of this extension is not localized. Rather, it has a system-wide impact. This we call AOP at the architectural level. The knowledge of how and where the system has to be extended or adapted can be captured in a FS graph (see Figure 8). Observe the effect of requiring broker functionality. The selection links stemming from the brokering functional requirement not only select the broker component to be placed in the server, but also a message delivery component for placement in the client. Effectively, the FS graph ensures that all refinements and extensions to satisfy a particular requirement are effectuated.

5 Related Work

Architecture-Based Architectural Styles (ABASs) are proposed in [7] as a means to capture structural and behavioral aspects of (partial) design solutions together with their quality properties. An ABAS combines an architectural style with certain quality attributes. We do essentially the same in our FS-graph, but we have a stronger focus on a construction/generation-oriented representation of the architectural knowledge.

In goal-oriented requirements engineering, the relation between goals and requirements is represented explicitly [8, 13]. A natural continuation of this line of thought is to connect requirements with high-level design (i.e., architecture). This is done in GRL-UCM [9]. The work on Goal-Oriented Language (GRL) in combination with UCM (GRL-UCM) has a lot of similarities with the concepts presented in this paper. As in our approach, a link is established between requirements (AND-OR decomposition, soft goals, and tasks) and design solution in the form of UCM scenarios. Also a notion of refinement with UCM stubs and plug-ins is supported. Our approach differs in the following respects:

A plug-in for brokering:



The resulting Client-Server system:

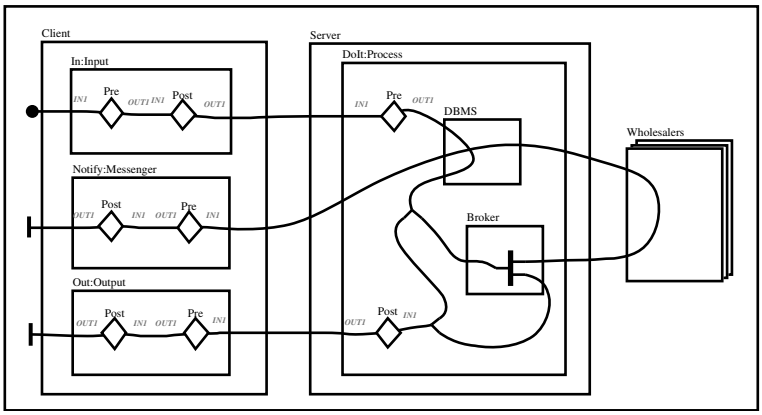


Figure 7: Extending the Client-Server system with broker functionality.

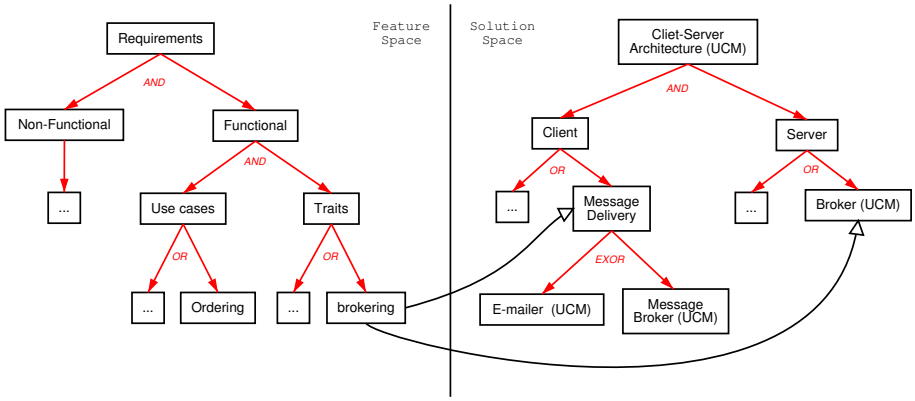


Figure 8: Feature-Solution graph for the Client-Server system.

- The explicit distinction between feature and solution space and the connections between them. This opens the way to recursively apply this idea. For instance, we can establish linkages between requirements (F-space) and software architecture (S-space). By the same token, we can establish linkages between archi-

tecture and detailed design solutions. But now, the software architecture level is the F-space w.r.t. to the detailed design level (S-space). The detailed design space in its turn can act as a F-space for the implementation (S-space).

- The S-space may contain nodes that are not connected to nodes in the F-space. This can be seen as degrees of freedom with which design alternatives can be explored.
- The idea of refinement can be applied recursively (i.e., plug-ins provide stubs). The GRL-UCM supports, on the other hand, one level of refinement only.
- We support a more advanced refinement mechanism in the form of multiple variability points. This allows us to use refine patterns that are not necessarily restricted to one, local point. The Observer pattern is a typical example to support this point.

The concept of refinement has been applied for many years, especially in the field of formal specifications, e.g., VDM and Z. A well-known approach to refinement in a component-oriented context is Catalysis [5]. The main idea in Catalysis is the concept of collaboration, which is a set of related actions involving multiple objects and resulting in a goal. A collaboration can be seen as a formalization of a use case. It is specified formally in terms of pre- and post-condition, and invariants using the Object Constraint Language (OCL). A collaboration is first described at a high level of abstraction. This description serves as a formal specification to be refined in subsequent steps. Although refinement is a key concept in Catalysis, it does not support the notion of variability points. That is, no clues are given whatsoever on where and how to refine. In our approach, refinement is done automatically on the basis of type and structure matching and the content of the FS graph. Also, no attention is paid to non-functional quality attributes, like flexibility, security and performance. The main emphasis is on behavioral refinement.

Surprisingly, refinement is not a main issue in most Architectural Description Languages (ADL) (see [11] for a classification framework and a survey of ADLs). Most ADLs favor bottom-up composition in the sense that components are treated as black-box, building blocks. SADL [12] and Rapide [10] are noticeable exceptions.

SADL supports a method for stepwise refinement of an abstract architecture into a lower level architecture. A refinement pattern is applied in each refinement step and maps the architectural style (e.g., dataflow, pipes and filters, implicit invocation) of the abstract architecture to the architectural style of the refined architecture. These refinement patterns are defined and proved correct independent of a particular architecture. SADL mappings have a rather global nature, whereas in our approach we can handle local as well as global refinements guided by the FS graph.

Rapide is a concurrent event-based simulation language for defining and simulating the behavior of architectures. The underlying semantical model of Rapide is based on partially ordered event sets (posets). An abstract and a refined architecture can be related by mapping concrete events to abstract events. These mappings provide the means for comparative simulations of architectures at different abstraction levels to check whether a refined architecture is in conformance with the abstract architecture. It could be interesting to explore this idea in our approach, after all, UCMs do imply ordering of events too. However, this is a subject that has currently not our focus of attention.

The idea of Pre- and Post-stubs has been derived from composition filters used in the programming language SINA [14]. The composition filter model consists of input and output filters that surround an object and affect the messages sent to and received by that object. Composition filters can be seen as objects in the role of proxies that perform additional pre- and post-processing.

6 Concluding Remarks

We have discussed an approach for top-down composition of software architectures. It is centered around a Feature-Solution (FS) graph in which requirements can be linked to design solutions and alternatives. The design solutions are expressed in UCM, which provides an intuitive way of showing how things work globally. UCM is not a prerequisite for our approach, though. The idea of supporting multiple variability points and plug-ins in combination with a FS graph can also be used with modeling techniques like UML, or even with a text-based specification language.

Our FS-graph serves two purposes. Firstly, it contains knowledge for solving problems in a particular domain. Thus viewed, the graph can be used to guide an iterative, quality-driven process to software architecture generation.

In this iterative process, the abstraction level is successively lowered because of refinements made. Secondly, the FS graph can be used to make system-wide adaptations. This ensures a consistent, system-wide application of refinements. These system-wide refinements resemble aspect-oriented programming, albeit applied at a much higher level of abstraction.

We are currently using the FS-graph to codify the architectural knowledge from various real-life architecture projects. Our experiences indicate some promising areas of further research:

- The version of the FS-graph discussed in this paper only contains product knowledge. We are currently investigating the incorporation of process knowledge in the FS-graph as well. When selected, such process nodes then guide the the architect in finding appropriate design solutions. The process steps encoded in process nodes can be based on heuristics, for instance, in the form of production rules (if $\ll$a particular situation X is encountered$\gg$ then $\ll$select solution Y$\gg$). Alternatively, process nodes may alert the designer that certain design choices have to be made. These small process steps can be embedded in the larger process cycles defined in an quality-driven, iterative approach. This leads to a content-driven refinement process.
- The relationships currently captured in the FS-graph are rather strong: "when high flexibility is required, select the Observer pattern". Actual relationships between requirements and solution chunks are often of a much weaker kind: "when high flexibility is required, we suggest to apply the Observer pattern". To represent these weaker links, the FS-graph will have to be enriched with additional kinds of relationships which, in turn, require additional resolution mechanisms to get from (quality) requirements to acceptable architectural solutions.

References

[1] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[2] R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.

[3] R.J.A. Buhr and R.S. Casselman. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[4] Hans de Bruin and Hans van Vliet. Scenario-based generation and evaluation of software architectures. In Jan Bosch, editor, *Proceedings of the Third Symposium on Generative and Component-Based Software Engineering (GCSE'2001), Erfurt, Germany*, volume 2186 of *Lecture Notes in Computer Science (LNCS)*, pages 128–139, Berlin, Germany, September 10–13, 2001. Springer-Verlag.

[5] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1998.

[6] Gregor Kiczales, John Lamping, Anurg Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In M. Askit and M. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97), Finland*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242, Berlin, Germany, June 9–13, 1997. Springer-Verlag.

[7] M.H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architectural styles. In P. Donohue, editor, *Software Architecture*, pages 225–244. Kluwer Academic Publishers, 1999.

[8] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Conference Proceedings ICSE'00*, pages 5–19, Limerick, Ireland, 2000. ACM.

[9] Lin Liu and Eric Yu. From requirements to architectural design: Using goals and scenarios. In *ICSE'2001 Workshop 9, From Software Requirements to Architectures (STRAW'2001)*, pages 22–30, Toronto, Ontario, Canada, 2001. ACM.

[10] David Luckhalm and James Vera. An event-based architectural description language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[11] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[12] Mark Moriconi, Xiaolei Qian, and R.M. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[13] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaiqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, January 2001.

[14] TRESE project. WWW: http://trese.cs.utwente.nl/sina/.

A A Brief Introduction to Use Case Maps

A UCM is a visual notation for humans to use to understand the behavior of a system at a high level of abstraction. It is a scenario-based approach showing cause-effects by traveling over paths through a system. UCMs do not have clearly defined semantics, their strong point is to show how things work globally.

The basic UCM notation is very simple. It consists of three basic elements: responsibilities, paths and components. A simple UCM exemplifying the basic elements is shown in Figure 9. A path is executed as a result of the receipt of an external stimulus. Imagine that an execution pointer is now placed on the start position. Next, the pointer is moved along the path thereby entering and leaving components, and touching responsibility points. A responsibility point represents a place where the state of a system is affected or interrogated. The effect of touching a responsibility point is not defined since the concept of state is not part of UCM. Typically, the effects are described in natural language. Finally, the end position is reached and the pointer is removed from the diagram. A UCM is concurrency neutral, that is, a UCM does not prescribe the number of threads associated with a path. By the same token, nothing is said about the transfer of control or data when a pointer leaves one component and (re-)enters another one. The only thing that is guaranteed is the causal ordering of executing responsibility points along a path. However, this is not necessarily a temporal ordering, the execution of a responsibility point may overlap with the execution of subsequent responsibility points.

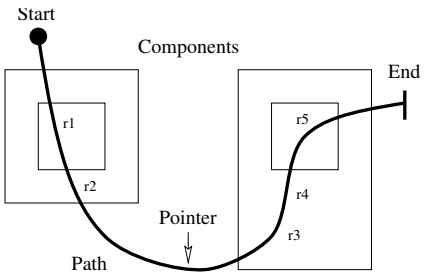


Figure 9: UCM basic elements.

A more realistic example is shown in Figure 10 depicting a distributed client-server system. Because the client communicates with the server over a network that can fail occasionally, a proxy server is included to provide transparent access to the real server. The proxy server is modeled as a stub for which two implementations are iven: a transparent proxy server which passes the requests to and the replies from the server unaltered thereby denying the possibility of network failures, and a proxy server with a timeout facility with which failures are detected. The notation used in the figure is supposed to be self-explanatory.

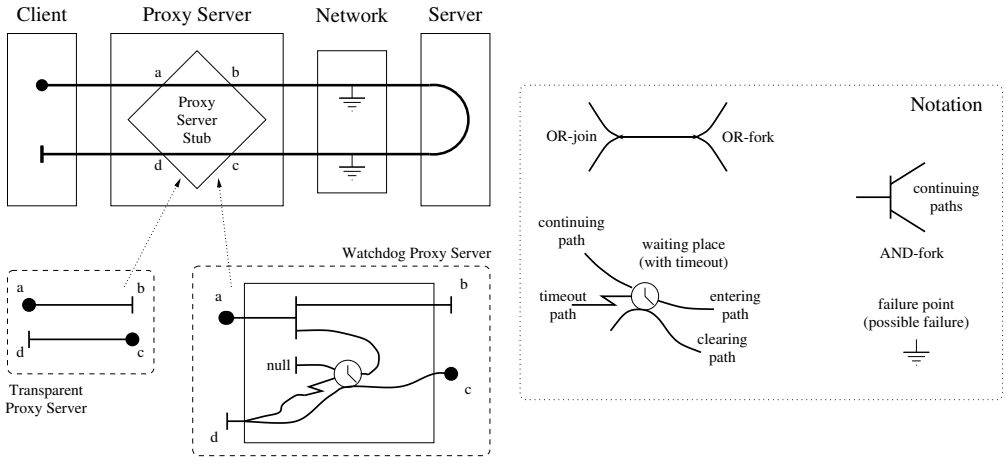


Figure 10: Distributed Client-Server UCM.